DOCUMENT RESUME

ED 134 150 IR 004 309

AUTHOR Worland, Peter B.
TITLE Teaching Structured Fortran without Structured Extensions.
INSTITUTION Gustavus Adolphus Coll., St. Peter, Minn.
PUB DATE 76
NOTE 25p.

EDRS PRICE MF-$0.83 HC-$1.67 Plus Postage.
DESCRIPTORS Algorithms; College Programs; *Programing; *Programing Languages; *Teaching Techniques
IDENTIFIERS *FORTRAN

ABSTRACT

Six control structures are used in teaching a college Fortran programing course: (1) simple sequences of instruction without any control statement, (2) IF-THEN selection, (3) IF-THEN-ELSE selection, (4) definite loop, (5) indefinite loop, and (6) generalized IF-THEN-ELSE case structure. Outlines, instead of flowcharts, are employed for algorithm development. Comparisons of student performance in structured and standard Fortran classes show no statistical significance, but suggest that the former approach is appropriate. (SC)

TEACHING STRUCTURED FORTRAN WITHOUT STRUCTURED EXTENSIONS

by

PETER B. WORLAND

GUSTAVUS ADOLPHUS COLLEGE

ST. PETER, MINNESOTA 56082

I should first define, or at least give some characteristics of what I mean by structured programming (S.P.). The main characteristic of S.P., as I see it, is that it embodies a "top-down" approach to the development of programs. That is, in the development of an algorithm one procedes from the general to the particular. The general statement of a problem is subjected to repeated decomposition into more detailed statements about how the problem is to be solved. To quote Niklaus Wirth: "The process of refinement continues until a level is reached that can be understood by a computer, be it a high-level programming language, FORTRAN, or some machine code." [1]

The result is - ideally - programs that are relatively easily understood and modified, programs that are reliable, not just efficient. And there is a good deal of evidence to indicate this is so; consider, for example, the article by F. T. Baker on the chief programmer team concept [2]. This approach is becoming more and more important; costs are increasing because of increasing salaries of programmers, and computation time is getting cheaper.

But why S.P. in FORTRAN? Wirth writes that S.P. is not possible in an unstructured language such as FORTRAN [1]. He writes further,

> "What is possible, however, is structured programming in a 'higher level' language and subsequence hand-translation into the unstructured language ...although this approach may be practicable with the almost superhuman discipline of a compiler, it is highly unsuited for teaching programming ...although there may be valid economic reasons for learning coding in, say, FORTRAN, the use of an unstructured language to teach programming - as the art of systematically developing algorithms - can no longer be defended in the context of computer science education."

Well, with respect to his comments on the systematic development of algorithms, I couldn't agree more. I do not ask my students to develop their algorithms in terms of FORTRAN. I do, however, require that they carry out the "hand-translation" into FORTRAN. This has not proved to be an "almost superhuman effort", the students manage quite well, at least on the problems I present them with. But, more on this point later.

Clearly, a language such as PASCAL with its rich structuring facilities, including those for data as well as instructions, is ideal. But there remain several good reasons for using FORTRAN. First, there is the advantage of availability (or rather, the unavailability of PASCAL or ALGOL or PL/1, etc.). There are reasonably compatible FORTRAN versions available from computers as powerful as an IBM 370/165 down to the smallest PDP 8. Like the proverbial mountain, FORTRAN is used because it is there (or perhaps because it was there). On larger systems such as the one I have the opportunity to use (a UNIVAC 1106), the choice of languages includes assembler, COBOL, two versions of FORTRAN, BASIC, RPG, SNOBOL, and ALGOL. "Aha," you say, "use ALGOL".

Use ALGOL, with its awkward, tedious, input-output statements; ALGOL that I am only weakly acquainted with, uncomfortable with, unlike the FORTRAN I know so well, that I've worked with so long (I'm a numerical analyst at heart); ALGOL, for which there are so few elementary textbooks that I need to deal adequately with the wide range of skills and talents I find in my students; an ALGOL that gives me puzzling results or error messages that strongly indicate compiler bugs -- bugs that when shown to the systems programmer, lead him to respond with something like, "Oh yeah. That looks serious. I'll have to look into that. I'll get back to you later." When in frustration, you explain the problem to the manufacturer's representative, his response is, "AL - who? Well, yes, that looks serious. We'll look into it and get back to you later." Much later. The point is that it's sometimes hard to get a language other than BASIC, FORTRAN, or COBOL that's adequately supported.

Another reason for using FORTRAN is related to the latter point; it's the matter of a student finding a job when he/she graduates. I have known quite a few students who have landed career-path jobs simply bacause they had some experience writing FORTRAN, or, even better, COBOL, programs. With some exceptions, a student competing for such positions, with a knowledge of PASCAL or ALGOL instead of FORTRAN or COBOL will not get the job.

A third advantage of languages like BASIC and FORTRAN, although many will consider this to be weak, lies in their relative simplicity. Although I have no hard data to back this statement up, I believe that it is easier for students to learn to program with such languages than with ALGOL or PL/1. If some basic concepts of control structures can be preserved using FORTRAN, I believe it's worthwhile. Because it is simpler, more students can learn to write good programs. The interested students can learn PASCAL or ALGOL as a second language. However, at a college of our size the resources are not available for teaching a second language.

Most of the ideas presented here are not new. It was Hull, I think, that first suggested the idea of structured FORTRAN in [3]. (You might know the idea would start with a numerical analyst). A flurry of papers on the subject of S.P. and structured FORTRAN appeared shortly after that. The paper by Meissner [4] has a large collection of references on the subject. In all the papers I've read on structured FORTRAN, the authors recommend the use of a pseudo-language in which a structured algorithm may be written and then translated by a preprocessor into FORTRAN. The same paper by Meissner [4] contains a list of such preprocessors.

This fall I'm teaching, for the second time, an Introduction to Computer Science course using FORTRAN with concepts of structured programming, but without structured extensions. Why, you may ask, would I do something as silly as that? Well, in my case it was easy. I didn't have time, nor did I have the money. By the time I had decided that this approach was a good idea, the semester was about to begin. Also, other jobs took precedence between the previous and current semester. There was no time for me to write my own preprocessor and the school doesn't have the money to buy one.

What other reasons could there be for not using such extensions? Ideally, the structuring facilities should be built into the compiler. This is already happening, and my approach may soon be obsolete. The FORTRAN Standards Committee is producing a FORTRAN version with a number of new control structures. Of course, we will still have to wait until the computer manufacturers announce these new versions.

Until then we must rely on a preprocessor, which accepts "structured FORTRAN" and produces standard FORTRAN as output to a file which must be executed later as a FORTRAN program. This is in itself an awkward business; the student is running two programs instead of one. To debug such a program he/she may have to examine the output from the preprocessor as well as the FORTRAN compiler. For very small systems (e.g., an 8K PDP 8) this approach may not be possible at all.

Of course, if the student is required to account for the conversion of "structured FORTRAN" into standard FORTRAN, these problems do not arise. I will point out a few other advantages a bit later.

In my approach to structured FORTRAN, I use essentially six control structures. These are all based on the ideas of Hull [3], and on the improvements suggested by Charmonman and Wagener [5]. These structures are presented to the class in the order given here, as soon as the class has been introduced to the idea of conditional transfer.

The first, and most elementary structure, is the simple sequence, which consists of a sequence of instructions without any transfer of control statements. For example,

```
      .
      .
      .
READ (5,100) A,B,C
X= (A+B+C) / 3.0
Y= SQRT (X)
WRITE (6,200)  X,Y
      .
      .
      .
```

is such a sequence. This sequence could be delimited by BEGIN and END comments, as Hull suggests, to give the appearance of a block structure, but I have found that they are not really necessary with this approach. The fact that such sequ-

ences always contain, or are nested within, other structures is sufficient to delimit them.

The next two structures are the ones most difficult to deal with using this approach. They are the IF-THEN and IF-THEN-ELSE structures, sometimes called selection structures. The general forms and examples of both are the following:

```
                IF(condition)                          Go To m
                                                       Go To n
C               THEN  (optional comment)
m                [do this if the condition is true]
C               END IF

n               ...
```

```
                IF  (X. LE. 50)                        Go To 40
                                                       Go To 50
C                THEN DOUBLE THE WEIGHTING FACTOR
40                 W (I) = 2.0 * W(I)
                   TSCR (I) = TSCR (I) + W(I) *X
                   I = I + 1
C               END IF

50              ...
```

Figure 1: The IF-THEN structure with an Example

```
      IF (condition)                                    Go To m
                                                        Go To n
C           THEN  (optional comment)
m
              [ do this if the condition is true ]

                                                        Go To p
C           ELSE  (optional comment)
n
              [ do this if the condition is false ]

C     END IF
p     ...
```

```
      IF (NAME .  EQ.  MNAME)                           Go To 10
                                                        Go To 20
C           THEN  FOUND  FORMER  EMPLOYEE
10            WRITE (6,300)  NAME, NUM, TAX
              TTAX = TTAX + TAX
              SW = 1
                                                        Go To 30
C           ELSE  CONTINUE  SEARCH
20            I = I + 1
              MNAME = NAMEMP (I)
              SW = -1
C     END IF
30    ...
```

Figure 2: The IF-THEN-ELSE Structure with an example

In both cases the box is given for emphasis. The structure in the boxes look very much like the corresponding construction in ALGOL or PL/1. And that is the idea; the students are told to write this part of the structure first and then to fill in the necessary GO TO's and statement numbers to make it work the way it looks. Once the students get the idea, it becomes a trivial bookkeeping step to put in the necessary GO TO's and statement numbers.

I do emphasize that the GO TO's should be used only to implement the control structures given here, if possible (like Dijkstra, I am not dogmatic about GO TO's). Also, note that the GO TO's are placed as far to the right as possible to improve the readability of the programs.

A nice feature of this approach is the ability to use comments with the THEN and ELSE keywords, as in the above examples. This feature is important in the formation of indefinite loops, to be discussed next.

Definite loops - those whose termination depends on a counter of some sort - are easily handled using DO loops. The only special requirement that I have is that the students terminate each DO loop with a CONTINUE statement to clearly mark the end of the loop.

Indefinite loops - those whose termination depends on some parameter or condition - are another matter. They are, to my mind, the least attractive of the control structure I use. What I am trying to do, of course, is to stimulate a DO - WHILE or DO - UNTIL kind of control structure. The general form, together with an example are:

```
C       LOOP  (optional comment)
m        ...
         ...
C       ***EXIT FROM LOOP  (optional comment)
           IF (condition)  Go To p
           ...                                  Go To m
C       END LOOP
p        ...
```

```
C       LOOP WHILE SCORE > = 0
10         READ (5,100)  NAME, SCORE
C       ***EXIT FROM LOOP WHEN END OF LIST
           IF (SCORE .LT.0.0) Go To 20
           SUM = SUM + SCORE
           SUM SQ = SUMSQ + SCORE * SCORE
                                                  Go To 10
C          END LOOP
20         ...
```

Figure 3: An Indefinite Loop Structure with an Example

Note the effective use of the comments following the keyword LOOP. This is an improvement in the schemes given by Hull, and later by Charmonman and Wagener. Again, a single GO TO is used to implement the loop. The awkwardness is in the exit from the loop. An IF statement must be used, and it should be clearly labeled as an exit from the loop. A comment can also be used here to indicate the reason for termination.

The last control structure that the students are exposed to is the very useful CASE structure, which is a generalization of the IF-THEN-ELSE construction. The general form, together with an example is:

```
C       GO TO (M1, M2,...Mk), v
M1
           CASE 1:  (optional comment)
                [do this if v=M1]                    Go To P
C          CASE 2:  (optional comment)
M2
                [do this if v=M2]
                                                     Go To P
                   ...
C          CASE K:  (optional comment)
Mk
                [do this if v=Mk]
C       END CASES

P              ...
```

```
        GO TO (50, 60, 70), IX
C          CASE 1:  X <= -1
50
                Y=X+2                                Go to 100
C          CASE 2:  -1<X<1
60
                Y = X**2
                                                     Go to 100
C          CASE 3:  X >= 1
70
                Y = 3*X + 1
C       END CASES
100
```

Figure 4: CASE Structure and an Example

Once again, appropriate comments can be very effective. The key to the use of the CASE structure is in the determination of the values of the integer variable v, and in the fact that the number of statement numbers in the GO TO list can be more than the number of cases if some of the numbers are repeated. I give my students a number of examples and exercises which force the student to

do some thinking about particular ways to use the CASE structure effectively. For example, one would not want to use a complex set of nested IF-THEN-ELSE structure simply to define v; in that case the CASE structure might as well be eliminated altogether.

You will note how similar these control structures appear to those used in PL/1 and ALGOL. Structured programs, no matter what the language, look very similar.

The students are exposed to all of these control structures as early as possible - usually by the end of five weeks. Naturally, this approach is not without its problems. Some of the typical - but early - errors are illustrated in the next three figures:

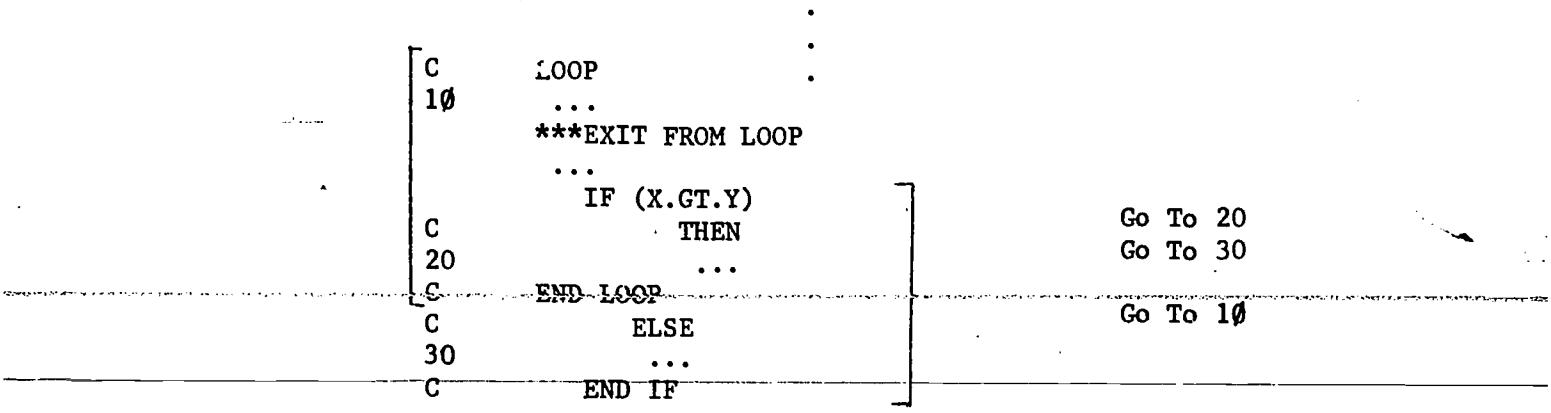

Figure 5: Overlapping Structures

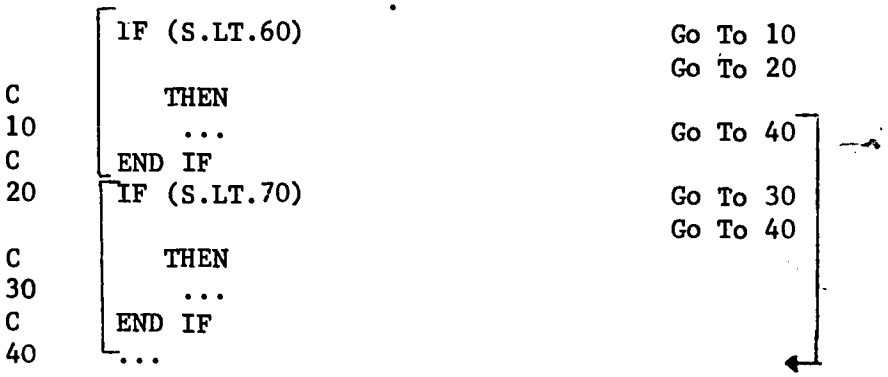

Figure 6: Jumping Over a Structure

```
IF (X.GT. COST)
   THEN
      IF (A-1.LT.0)
         THEN
            ...
         ELSE
            IF (P.NE.Q)
               THEN
                  ...
            ELSE
               ...
```

Figure 7: Missing END IF's

Figure 5 illustrates the case of overlapping structures. Some students seem to at first confuse the ideas of overlapping versus nested. When this happens, I give the student some helpful hints as to how the particular problem can be solved in a different way, without such a mess. Having seen this, the students do not make this kind of error again, at least in my experience.

Figure 6 illustrates how some students, in attempting to be efficient, jump past a block of code. Again I repeat to them the near-axiom: "never use a GO TO except to implement one of the control structures." With some hints the students realize that the problem can be handled in another way (e.g., a CASE structure or nested IF-THEN-ELSE structures), and this type of error usually does not arise again.

Figure 7 shows a common frustrating problem -- the omission of the END IF delimeters. The result is a sequence of code that looks ambiguous, especially with the last ELSE improperly indented. Sometimes the student is not even sure what he/she intended. In this case the student is reminded that each structure has a beginning and an end, and that an examination of his/her design of the program

will straighten out the ambiguity (more on this later). Remember that the idea is that I or anyone else should be able to read the code as a sequence of control structures, where control flows uniformly from top to bottom, without looking at the GO TO's at all.

In general, the students seem to get the idea fairly quickly. Putting in the GO TO's, comments, and line numbers become more or less automatic. One complaint I have seen of this approach is that it requires a "disciplined approach to programming". But, after all, isn't that precisely what the activity of programming needs? The errors described above could be made in PL/1 or ALGOL just as easily. The difference is that, rather than have the compiler detect the errors, I, as the instructor, must point them out. The extra burden is on me, but not so much on the students.

Of course, I have thus far discussed only the control structures that I need in the code. And S.P. is concerned not only with the control structures, but also with the whole design process. The process of algorithm design should really be language independent, except for the last "refinement", to use Wirth's terminology. Refinement refers to the step-by-step inclusion of more detail in the description of the problem solution.

In general, I try to get the students to develop their programs from the top to the bottom (i.e., from the general to the particular). Furthermore, I try to get them to "think modularly", that is, to break up large problems into sequences of small, easy-to-handle modules, which arise in a natural fashion in the process of refining the solution. For this aspect of the course I teach by example (see below).

Rather than use flowcharts to do program design, I prefer to use outlines. To me, flowcharts are outmoded, clumsy, and take too much time to construct. A well-done flowchart may look attractive but it does not look like a program. There remain considerable hurdles before a complex flowchart is converted to an equivalent program. I can best illustrate what I mean with the following example,

which I discuss in class.

The problem is to write a program that will find the median of a set of test scores. I find that this is a good illustration for the use of arrays. After some discussion of how we would find the median if we did the problem by hand, I write down the following outline of the solution:

I. Read all the numbers into an array, SCORE.

II. Sort the elements of SCORE into ascending order.

III. Compute and print the median.

Next, we decide that each of these steps is complex enough to be refined independently. We note that step I. is not difficult since we had encountered similar problems before. We decide to use a negative "score" as a trailer. Refining step I., we have:

I'. Initialize the loop index I to 1.

II'. Repeat the following until score SCORE (I) < 0:

- A. Read a score into array location SCORE (I).
- B. Add 1 to I.

III'. Set N = I - 1, the total number of scores.

That part of the problem is reasonably refined; it would be a simple matter to code the later outline into structured FORTRAN. Therefore, next we concentrate on step II. Since sorting had not been discussed before, I spend some time with some short samples of data considering how we might sort them by hand. I lead them into a simple exchange sort algorithm. The following outline is the result:

IV. Set NOC = N - 1, the numbers of comparisons to be made.

V. Repeat until switch W does not change value:

A. Set switch W=0

B. For I=1 to NOC

1. If SCORE(I) > SCORE(I+1)

then

a. Interchange SCORE(I) and SCORE(I+1)

b. Set W=1

C. Exit from loop if W=Ø

Next, we discuss how to actually implement an interchange. Having done so, we refine the latter outline to the following:

IV. Set NOC = N-1, the number of comparisons

V. Repeat until switch W does not change value:

A. Set W=0

B. For I=1 to NOC

1. If SCORE(I) > SCORE(I+1)

then

a. Set TEMP = SCORE (I)

b. Set S(I) = S(I+1)

c. Set S(I+1) = TEMP

d. Set W=1

C. Exit from loop if W=0

Next we consider the problem of computing the median, now that the scores have been sorted. After some trials on some short samples of data, we determine that:

```
VI.  If N is odd
       then
          A.  the median is SCORE (N/2 + 1)
       else
          B.  the median is (SCORE (N/2) + SCORE (N/2 + 1))/2
```

Another refinement is needed to specify the determination of the odd-even question:

```
VI.  Set I = N/2
VII. If N = 2*I
       then (N is even)
          A.  the median is (SCORE (I) + SCORE (I+1))/2
       else  (N is odd)
          B.  then median is SCORE (I+1)
```

We note at this point that each of the outlines have been refined enough, and we can easily code the following program (introductory documentation is omitted here):

```
            DIMENSION SCORE (100)
            I=1
C           LOOP UNTIL NO MORE SCORES:
5              READ (5,100) SCORE (I)
            ***EXIT FROM LOOP IF TRAILER ENCOUNTERED
               IF (SCORE(I).LT.0)  GO TO 10
               I = I+1
                                                          GO TO 5
C           END LOOP
10          N = I-1
            NOC = N-1
C
C      SORT THE SCORES INTO ASCENDING ORDER
C
C           LOOP UNTIL SWITCH W DOES NOT CHANGE VALUE
12            W = 0
              DO 20 I = 1, NOC
                 IF (SCORE(I).GT.SCORE (I+1))             GO TO 15
                                                          GO TO 20

C                   THEN
15                     TEMP = S(I)
                       S(I) = S(I+1)
                       S(I+1) = TEMP
                       W = 1
C                 END IF
20            CONTINUE
C           ***EXIT FROM LOOP IF W STILL 0.
               IF (W.NE.0) GO TO 12
C           END LOOP
C
C      COMPUTE THE MEDIAN
C
                  I = N/2
                  IF (N.EQ.2*I)                           GO TO 30
                                                          GO TO 40
C                    THEN (N IS EVEN)
30                      MED = (SCORE (I) + SCORE (I+1))/2
                                                          GO TO 50
C                    ELSE (N IS ODD)
40                      MED = SCORE (I+1)
C                 END IF
50                WRITE (6,100) MED
                  STOP
                  END
```

Thus, in the design stages of a program, I ask the students to use outlines like those above, rather than flowcharts. From the beginning stage of the development, the problem solution is in outline form. The necessary variable names are introduced (and defined) in the outlines where convenient. More and

more details are introduced in successive stages; some of the steps in a given outline are treated as separate modules if they appear to be complex enough to consider by thenselves. I don't require the students to use a specific macro language in their outlines. I do stress, however, that the language used in the outlines should resemble the structures discussed above. This is reinforced by examples. Eventually, then, the outline, through successive refinement, begins to resemble structured FORTRAN. And each refinement, it seems to me, is more natural, more straightforward than with flowcharts.

Now, of course, this approach isn't really new either. I'm sure Nicklaus Wirth subscribes to it (or, at least, he has no aversion to flow charts) although I have not read a reference of his to that effect. I can recall that Daniel McCracken, at a conference some time ago, had said that he didn't like flowcharts, but he wasn't sure (at that time) what to replace them with.

And yet, with all that's been published on S.P., I'm amazed at the number of introductory textbooks on programming that still attempt to teach programming in the same old way. Students are never really taught how to write programs. Detailed presentations of the particular language elements are made. Many examples of programs and many exercises are provided. Sometimes flowcharts are shown, but with little emphasis on how to design one. I suppose that's why I'm writing this paper; I like to think of myself "spreading the word as a disciple of S.P.". The paper is written in the first person because it is one man's opinion; I don't expect to persuade everyone that my views are correct. However, I should note that there is at least one book on S.P. in FORTRAN, by Lynch and Rice, but I haven't seen it yet.

I would like to exhibit several more examples, these from students using this approach. Early in the course, even before the concept of looping is in-

troduced, I give my first exam. In that exam the students are asked to write a FORTRAN program to find the middle value of there numbers (i.e., the median) read from a data card. This is a difficult program to write on a test at this point. The following is an example written by a student using S.P. (admittedly, one of the better students). The second example was written by a student (also one of the better students) in my class where S.P. was not used. The examples speak for themselves.

```
100      FORMAT ( )
         READ (5,100) A, B, C
         IF (A.LE. B)                                  GO TO 200
                                                       GO TO 300
C           THEN
  200          X1=A
               X2=B
                                                       GO TO 350
C           ELSE
  300          X1=B
               X2=A
C        END IF
  350    IF (C.GE.X2)                                  GO TO 400
                                                       GO TO 500
C           THEN
  400          WRITE (6,100) X2
                                                       GO TO 700
C           ELSE
  500          IF (C.GE. X1)                          GO TO 600
                                                       GO TO 650
C                 THEN
  600                WRITE (6,100) C
                                                       GO TO 700
C                 ELSE
  650                WRITE (6,100) X1
C              END IF
C        END IF
  700    STOP
         END
```

Figure 8: Finding the Median of Three Numbers with S.P.

```
      READ (5,100) A,B,C
100   FORMAT ( )
      IF (A. GE.B)  GO TO 10
      IF (B. GE.C)  GO TO 20
      IF (B. GE.A)  GO TO 30
 70   WRITE (6,100) A
      GO TO 50
 10   IF (A. GE.C)  GO TO 60
      GO TO 70
 60   IF (B. GE.C)  GO TO 80
 90   WRITE (6,100) C
      GO TO 50
 80   WRITE (6,100) B
      GO TO 50
 20   IF (A. GE.C)  GO TO 70
      GO TO 90
 30   IF (B. GE.C)  GO TO 110
      GO TO 80
110   IF (C. GE.A)  GO TO 90
      GO TO 70
 50   STOP
      END
```

Figure 9: Finding the Median of Three Numbers Without S.P.

I'm proud of the next example - The Eight Queens Problem - which one of my students (admittedly one is very sharp) wrote just a week ago, after only six weeks of class. I haven't had time to check it carefully but it does appear to be correct, and it certainly is well structured, from a FORTRAN point of view at least.

```
C     'Eight Queens' by Barry Johnson.  10/20/76
C     Finds all possible ways to place 8 queens on a chess board
C     so that none may take any other.
C     The columns in the output represent the columns the
C     queens would be in, and the numbers printed out are
C     the respective rows they are in.

      INTEGER ROW, COLUMN, ATTACK, QUEEN(8), I
      WRITE(6,102)
      WRITE(6,103)
      WRITE(6,104)
      COLUMN = 1
      TEST = 'BEGIN'
C     START LOOP 1
C     START LOOP 2  **********
20    IF(TEST = 'BEGIN') ROW = 1
      TEST = 'BEGIN'
C        START LOOP 3 **********
10          CALL ATTACK
            ***EXIT FROM LOOP 3
```

```
            IF (ATTACK.NE.'YES')   GO TO 1
            ROW = ROW + 1
            CALL CHECK
                                                          GO TO 10
         END LOOP 3 **********
    1    QUEEN(COLUMN) = ROW
         COLUMN = COLUMN + 1
         ***EXIT FROM LOOP 2
         IF(COLUMN.LE.8)  GO TO 20
    C  END LOOP 2  **********
      WRITE (6,101) (QUEEN (I),I = 1,8)
      ROW = ROW + 1
      CALL CHECK
      TEST = 'MIDDLE'
                                                          GO TO 20
    C  END LOOP 1
  101  FORMAT (' ',8(I1,2X))
  102  FORMAT (' COLUMN:')
  103  FORMAT (' 1 2 3 4 5 6 7 8 ')
  104  FORMAT ('----------------')
    C  ****************************
    C  *    SUBROUTINE ATTACK     *
    C  ****************************
      SUBROUTINE ATTACK
      ATTACK = 'NO'
      IF(COLUMN.EQ.1)  RETURN
      N= COLUMN - 1
      DO 3 I = 1,N
         IF (QUEEN (I).EQ.ROW) ATTACK = 'YES'
         IF (ABS(FLOAT(COLUMN - I)/FLOAT(ROW-QUEEN(I))).EQ.1.0)
                              ATTACK = 'YES'
    3  CONTINUE
      RETURN
    C  ****************************
    C  *    SUBROUTINE CHECK      *
    C  ****************************
      SUBROUTINE CHECK
  100  FORMAT ('NO MORE SOLUTIONS')
    C  LOOP
    C  ***EXIT FROM LOOP
   30     IF (ROW.NE.9)  RETURN
    C  START LOOP
   30     IF (COLUMN .EQ. 1)                            GO TO 6
                                                        GO TO 7
    C        THEN
    6           WRITE (6,100)
      *********TERMINATE PROGRAM
                STOP
    C     END IF
    7     COLUMN = COLUMN - 1
          ROW = QUEEN ( COLUMN )
          ROW = ROW + 1
                                                        GO TO 30
    C  END LOOP
          END
```

The output has the form:

COLUMN:

1	2	3	4	5	6	7	8
1	5	8	6	3	7	2	4
1	6	8	3	7	4	2	5
.	.	.	.	.	.	.	.
8	4	1	3	6	2	7	5

NO MORE SOLUTIONS

The program given here has some minor structural improvements over the original, but no other changes. This student wrote the program in less than three days without any hints as to how to solve the problem. This program made 16,042 tests compared to Wirth's Pascal Program [9] that made 15,720 test to generate the 92 solutions. However, this program required 16 seconds CPU time on a UNIVAC 1106 compared to less than one for Wirth's program on a CDC 6400. But that's still not bad for a sophomore.

Overall, I think the results have been successful, given the constraints mentioned earlier. To summarize, I have a few statistics that indicate, to me at least, that teaching structured FORTRAN is certainly better than standard FORTRAN.

Table 1
A Comparison of Three Introductory Computer Science Courses

C1 = "non-S.P." in FORTRAN

C2, C3 = S.P. in FORTRAN

Statistics for the Entire Course (30 students)

Course	Mean	Std. dev.	T-values
C1	72.7	11.2	1.90
C2	78.5	12.5	

Statistics for the Final Exam (30 students)

Course	Mean	Std. dev.	T-values
C1	72.2	11.7	1.31
C2	76.5	13.5	

Statistics for the First Hour Exam (30 students)

C1	69.7	22.2	2.52 (C1 with C2)
C2	82.2	15.4	1.64 (C1 with C3)
C3	77.4	12.8	1.3 (C2 with C3)

The t-values for the entire course and for the first hours exam, (C1 with C2) and (C1 with C3), are significant at the 10% level. The t-value for the final exam is not significant. Also, it should be noted that, with respect to the program to find the median of three numbers given on the first test, only four students in class C1 wrote programs without any errors, compared to eight in class C2. These are not overwhelming statistics, but at least they are in favor of this approach to teaching programming.

Finally, I should point out that I am not dogmatic about this approach either. We are now writing our own preprocessor to handle structured FORTRAN. I am very interested to see how well my class will do using this preprocessor. I may change my mind, but I do not believe they will perform significantly better.

References

[1] N. Wirth, "On the Composition of Well-Structured Programs", ACM Computing Surveys, 6, 4 (1974), 247-260.

[2] F. T. Baker, "Chief Programmer Team Management of Production Programming", IBM Systems Journal, 11, 1 (1972), 56-73.

[3] T. E. Hull, "Would you Believe Structured FORTRAN?", SIGNUM Newsletter 8, 4 (1973), 13-16.

[4] L. P. Meissner, "On Extending FORTRAN Control Structures to Facilitate Structured Programming", SIGPLAN NOTICES, Sept., 1975, 19-29.

[5] S. Charmonman and J. L. Wagener, "On Structured Programming in FORTRAN", SIGNUM Newsletter 10, 1 (1975), 21-23.

[6] N. Wirth, "Program Development by Stepwise Refinement", Comm. ACM, 14, 4 (1971, 221-227.